Distributed Vertex Cover in Network Graphs

Abstract—Vertex cover, a minimal set of nodes to cover all edges in a graph, is an abstraction of coverage problems in sensor networks, transportation networks, etc, and is a well-known NP-hard problem. Minimum weighted vertex cover (MWVC) problem asks for further minimizing the cumulative weight of a vertex cover. We present new distributed k-hop algorithms for MWVC problem with theoretical and practical values. Our first 1-hop approximation algorithm, based on matching a maximal set of non-adjacent edges, is provably 2-optimal with a communication complexity of $O(\Delta)$. It compares very well with the current state-of-the-art in quality while significantly reducing communication cost.

We also explore an important variant, the problem of finding a series of vertex covers to maximize network lifetime. Our second algorithm, based on a key insight into the vertex cover problem of collecting partial covers from 2-hop neighbors, is an excellent practical algorithm. It is representative of a problem-structure based efficient sampling algorithm in the exponential size local solution space. We show that a partial cover based algorithm can be enhanced further to compete very well and exceed the lifetime obtained with state-of-the-art algorithms.

I. INTRODUCTION

The Minimum Vertex Cover problem and its weighted variant are NP-Complete problems with several known linear time sequential algorithms that provide constant approximations. The existence of such algorithms suggests that there is a constant time distributed algorithm that would provide a constant approximation for MVC or MWVC, but it has been shown that a constant approximation of MWVC cannot be found by a distributed algorithm in a constant number of rounds[9].

Here we present a distributed 2-optimal algorithm to solve MWVC in an expected running time of $O(\log n)$, based on the linear time sequential algorithm of Gonzalez[6]. This is not the first such algorithm to appear in the literature, but there are implementation advantages to our approach. In addition, we present an interesting subroutine that runs in constant time and improves the quality of solutions for both our algorithm and the prior algorithm of Koufogiannakis and Young[8]. This subroutine turns out to have practical value when applied to the related problem of sensor network lifetime.

Definitions and descriptions of prior work is presented in Section II. In Section III we describe our own distributed algorithms for both network lifetime and MWVC. Section IV contains the description of our simulation software design, which provides the engine for the experiments that are described in Section V.

II. BACKGROUND

A. Definitions

The coverage problems in this paper are common coverage problems which are known to be NP-Complete. For convenience, the problem definitions are provided here.

1) Minimum Vertex Cover: Given an undirected Graph $G(V, E)$, a Vertex Cover of $G$ is a set of vertices $V'$ such that for each edge $e_{u,v} \in E$, $u \in V'$ or $v \in V'$. The Minimum Vertex Cover Problem is to find the smallest possible vertex cover of $G$.

2) Minimum Weighted Vertex Cover: Given an undirected Graph $G(V, E)$, where each $v \in V$ has a positive weight $w(v)$, minimize $\sum_{v \in V'} w(v)$.

3) Network Lifetime: Given a sensor network $N(S, T)$, the Network Lifetime of $N$ is the maximum time that $\forall t \in T, \exists s \in S[s$ covers $t]$.

4) Model: All of the distributed algorithms described are assumed to be running on a message passing model, the compute nodes are mapped to the vertices of the graph, and the edges of the graph represent viable paths for communication between nodes.

B. Prior Work

Sequential Linear time algorithms for covering problems are surveyed in detail in [11]. The seminal paper on Linear Programming techniques for constant ratio approximation of MWVC was published by Bar-Yehuda and Even in 1981 [1]. Gonzalez created a 2-optimal LP-Free linear time algorithm based on Maximal Matching in 1995 which is the basis of our distributed algorithm [6].

We are aware of two distributed algorithms for minimum weighted vertex cover. A 2-optimal algorithm based on maximal matching was published in 2008 by Grandoni et. al [7]. We implement a simpler algorithm
presented by Koufaganis and Young in 2009 for performance comparison [8]. The Koufaganis/Young algorithm is described in detail in Section II-C.

The network lifetime problem is introduced in detail by Cardei et. al in [3]. A mathematical analysis of the problem is provided by Legakis et al in [10]. Brinza and Zelikovski’s deterministic algorithm [2], detailed in Section II-D is used in this paper as a point of comparison. The issue of communication costs is addressed by Zhao et. all, along with a formal definition for the Connected Target Coverage Problem [13]. Dhawan and Prasad introduce the use of Dependency Graphs for the network lifetime problem in [12], this work provides the jumping off point for the algorithm introduced in Section III-C. Dependency Graphs are explored further in Section II-E.

C. The Koufaganis/Young Algorithm

D. The DEEPS Algorithm

E. Dependency Graphs

III. ALGORITHMS

A. Distributed Generalized Maximal Matching Algorithm

1) Description: Algorithm 1 is our distributed implementation of the 2-optimal minimum weighted vertex cover algorithm presented by Gonzalez. [6] The Gonzalez algorithm proceeds by selecting each edge in turn and choosing one of the endpoints of that edge to add to the cover. The sequential algorithm goes through each edge in turn and assigns the edge a weight according to equation 1. If the weight of a vertex is equal to the sum of its incident edge weights, that vertex is added to the cover.

\[
w(e(u,v)) = \min \left\{ w(u) - \sum_{i \neq v} w(e(u,i)) \right\}, \quad w(v) - \sum_{i \neq u} w(e(i,v)) \quad (1)
\]

The distributed version of the algorithm chooses some disjoint set of edges and assigns weights as described. The precise method of choosing edges and updating weights is given in Algorithm 1. The automaton capturing the state transitions of Algorithm 1 is given succinctly in Figure 1. Each vertex begins in the C state, and chooses to either send invitations (I), or listen for invitation (L). Vertices in the I state choose one neighbor to send an invitation to and transition to a waiting state (W), and vertices in the L state choose one invitation to accept. The acceptance message is sent during the response state (R). All vertices then update their status (U). Vertices that have either chosen to join the cover or which have no undecided neighbors will transition to the done state, (D), and other vertices return to state C.

During the U state, vertex pairs formed during the invitation/acceptance phases are able to assign a weight to the edge between them independently using equation 1, and therefore decide whether or not to join the cover. During the E state neighboring vertices are able to update some of their own edge weights, by assigning a weight of zero to any edges incident to a vertex which has joined the cover.

2) Performance: The sequential version of Algorithm 1 has been shown to be 2-approximate for any ordering of edges. The distributed version can never select contiguous edges in a single round, as each node can form at most one partnership in a round. Because the simultaneous selection of vertexes to join the cover based on disjoint edges is exactly equivalent to the sequential selection of vertexes by selecting each edge under consideration in turn.\footnote{This algorithm could therefore also be used to provide an edge coloring in approximately the same time, equivalent to a greedy linear algorithm.}

Theorem 1. Algorithm 1 (DGMM) will always generate a 2-optimal cover in \(O(\Delta)\) communication rounds.

Note 1 (Mapping to Sequential Algorithm). DGMM is based on a sequential algorithm (GMM), which takes as an input a graph and produces a 2-optimal vertex cover of that graph. The sequential algorithm selects each edge of the graph in turn, in arbitrary order, and compares the endpoints of that edge. The edge is assigned a weight according to Equation 1. If one endpoint is already in
Algorithm 1  Distributed Maximal-Matching-based Minimum-Weighted Vertex Cover Algorithm (DGMM)

for all \( v_u \in V \) in parallel do
\( S_u \leftarrow False \) \( \triangleright u \) is not in the cover
state \( \leftarrow C \)
repeat
Broadcast \( S_u \)
if \( S_v = \text{True} \) for \( v_v \) incident to self then
Set Weight \( e_{u,v} \leftarrow 0 \)
end if
if state = \( C \) then
State \( \leftarrow (I \lor L) \) \( \triangleright \) Coin toss selects state
else if state = \( I \) then
Randomly select an unweighted edge, \( e_{u,v} \)
Broadcast \( I_v^u \) \( \triangleright u \) Invites \( v \) to weight \( e_{u,v} \)
state \( \leftarrow W \)
else if state = \( L \) then
Recieve \( I_x^y \) \( \triangleright \) all local invites
if \( y = u \) then \( \triangleright \) invite is targeted to \( v \)
store \( I_x^y \)
end if
State \( \leftarrow R \)
else if state = \( R \) then
Randomly Select \( I_x^y \) \( \triangleright \) from stored invites
Broadcast \( R_x^y \) \( \triangleright u \) accepts \( v's \) invitation
Update Weight \( e_{u,v} \) \( \triangleright \) by Equation 1
if \( \sum w_{e(i,u)} = w(u) \) then
\( S_u \leftarrow \text{true} \) \( \triangleright u \) joins the cover
state \( \leftarrow D \)
else
state \( \leftarrow C \)
end if
else if state = \( W \) then
Recieve \( R_x^y \) \( \triangleright \) all local responses
if \( y = u \) then \( \triangleright \) response is to \( v \)
Update Weight \( e_{u,v} \)
if \( \sum w_{e(i,u)} = w(u) \) then
\( S_u \leftarrow \text{true} \) \( \triangleright u \) joins the cover
state \( \leftarrow D \)
else
state \( \leftarrow C \)
end if
end if
end if
until \( S_u = \text{true} \) OR \( S_v = \text{true} \) \( \forall v_v \) incident \( v_u \)
end for

For DGMM, the Graph is represented as a network of compute nodes, with the nodes representing vertexes and connections between the nodes representing edges. When the algorithm terminates, every node in the network that is in the cover should be in an “on” state, and every node that is not should either be an “off” state.

Note 2 (Communication Model). As mentioned in Section II-A4, we assume a message passing model of distributed computing. In each communication round, it is assumed that every node can communicate with its neighbors. Communication is assumed to be synchronous and symmetric: if node \( a \) is a neighbor of node \( b \), node \( b \) is a neighbor of node \( a \), and if node \( a \) has counted \( x \) communication steps, so has node \( b \).

A “communication round” is actually three steps: an invitation sending step, an information response step, and an exchange step where neighbors share changes in status.\(^2\)

Note 3 (Local Information). At the beginning of each communication round, each node has a list of it’s neighbors, their current state, the edges associated with those neighbors, and the results of any previous computation performed on those edges.

Proof of Theorem 1:

Summary 1. We show first that our algorithm conducts the same steps in the same order as a sequential algorithm that is known to produce a 2-optimal result, next that the algorithm will terminate in \( O(\Delta) \) communication rounds.

Lemma 1.1. DGMM examines each edge once in a manner equivalent to GMM.

Proof of Lemma 1.1: \( \blacksquare \)

Lemma 1.2. DGMM will terminate in \( O(\Delta) \) communication rounds.

Proof of Lemma 1.2: In each communication round, roughly half of the nodes will choose to receive invitations. For each of these nodes, roughly half of their neighbors will be sending invitations, and the probability of receiving an invitation from one of these neighbors is \( \frac{1}{2} \) it should be noted that the K/Y algorithm also requires three communication events per round.
1 : \Delta. Taken together. As each node that receives one or more invitations will definitely respond to one invitation, in any given round, the probability of joining a node pair is 1 : \Delta. Of the two node pairs, one will weight a single edge, while the other will weight \Delta edges, each of which will also be updated for \Delta neighbors. This means that in a given round, roughly half of the nodes will make a decision for 1 of their edges, leading to a communications complexity of O(2\Delta).

Therefore, Theorem 1 is correct.

B. Redundancy Checking

When vertices make local decisions to join a cover, it is difficult to judge whether any neighbor will also decide to join the cover. In some cases, this leads to vertices joining the cover which can be subsequently removed while still retaining full coverage. Removing these nodes will certainly reduce the total weight of the cover. We therefore implement a redundancy checking algorithm. Figure 2 shows the progression of Algorithm 2.

![Fig. 2: Redundancy Checking Algorithm](image)

The redundancy checking algorithm proceeds step-wise, similar to Algorithm 1, and many of the same arguments apply. One difference is that redundancy checking is run a single time for each node, nodes check with their neighbors once and then decide to turn off only if they are the largest redundant node in their immediate neighborhood. Because no two neighboring nodes can both be the largest—assuming a tie breaking mechanism such as unique ids—such a decision cannot break the cover. Also, because nodes make the decision simultaneously and globally, the additional number of communication rounds required is constant.

The concept behind Algorithm 2 is simple; a vertex that does not have any edges covered by a vertex of

---

**Algorithm 2 Redundancy checking algorithm (RC)**

**Require:** G(V, E), w(v): A vertex-weighted graph

**Require:** state(v_i) = D \lor F  \triangleright See Fig 1

**Ensure:** state(v_i) = (I \lor O)

1: for all v_i \in V synchronously do
2: if state = D then
3: DO check-finished
4: else if state = F then
5: DO check-redundant
6: end if
7: end for
8: procedure CHECK-FINISHED
9: if \forall v_u incident v_i, state(v_u) = D \lor F then
10: if v_i \in C then \triangleright Where C is the Cover
11: if \forall v_u incident v_i, v_n \in C then
12: redundant \leftarrow TRUE
13: end if
14: state \leftarrow F
15: end if
16: else
17: state \leftarrow D
18: end if
19: end procedure
20: procedure CHECK-REDUNDANT
21: if \forall v_u incident v_i, state(v_u) = F \lor I \lor O) then
22: if redundant = TRUE then
23: if \forall v_u incident v_i, redundant(v_u) = FALSE then
24: remove v_u from C
25: else
26: a \leftarrow [w(v_i)] + [w(v_u)\forall v_u incident v_i]
27: if maximum of a = v_v then \triangleright v_v has the largest weight in its neighborhood
28: remove v_v from C
29: end if
30: end if
31: end if
32: state \leftarrow I \lor O
33: else
34: state \leftarrow F
35: end if
36: end procedure
greater weight removes itself from the cover. This simple idea provides some valuable results extending network lifetime without incurring large communication costs, as we see later in Section III-C2. When examining target coverage in a sensor network, most current algorithms ignore the communication cost of establishing the target cover[13]. One reason for this is that the cost is generally considered to be a constant, that is, any algorithm that provides continuous coverage must perform a global reshuffle periodically in order to maximize network lifetime.

C. Partial Cover Dependency Graph

Network Lifetime and Minimum Weighted Vertex Cover are both NP-Complete problems. It has also been proven that MWVC cannot be approximated to a constant factor locally within any constant number of communication rounds [9]. This limitation must apply to target coverage as well. We developed our algorithm continuously covering the edges for extending total network lifetime based on the Dependency Graph which provides an algorithmic framework for target coverage and related problems [4]. It has been shown to produce good results when applied to the Lifetime Maximization problem for target coverage in sensor networks [5].

The framework applies to problems where local solutions can be combined to form a feasible global solution. The essential steps of the framework are as follows.

1) Establish that combined local solutions lead to a feasible global solution.
2) Model the state space of the local solutions.
3) Determine a priority heuristic for local solutions.
4) Design a reasonable negotiating strategy between neighbors.

A detailed description of each of these steps can be found in [4].

The application of the framework relies on dependencies between local solutions. In the case of the vertex cover problem, there are several approaches that can be taken to determine what a local solution is. The simplest approach is to have each vertex only consider edges incident to itself. Naively, each vertex would have exactly two local solutions, the cover containing itself and the cover containing all of its neighbors. These two covers are node disjoint and lack any dependencies to prioritize meaningfully. Therefore, one may consider the vertex covers for edges incident to 1-hop neighbors as well. Now a large number of possible covers have to be considered. The number of possible local covers for a vertex of degree $\Delta$ is $\sum_{i=0}^{\Delta} \left( \frac{\Delta}{i} \right)$.

The number of local covers increases as a function of the density of the local neighborhood. If $\Delta$ is small, this is not a problem, but as $\Delta$ increases the number of potential local covers increases rapidly. The Partial Cover Algorithm samples this exponential space and reduces the number of solutions to $O(\Delta)$. A given vertex can only see two covers for it’s own edges: the cover containing itself, and the cover containing all of its neighbors. The partial cover algorithm samples the solution space based on what vertices would have to be on if either of these two covers were off.

1) Construction of the Partial Cover Dependency Graph (PCDG): Given a graph $G(V,E)$, for each vertex in $V$ we can define a partial cover dependency graph consisting of the partial cover pair $C_v, C_{n(v)}$ for $v$, and the partial cover pair for each neighbor of $v$. Given a node $v \in V$, $C_v$ consists of $v$ and its two-hop neighbors, while $C_{n(v)}$ consists of $v$’s one-hop neighbors. Two nodes are connected (dependent), if the covers are non-disjoint. For clarity, we define terms below.

**Definition 1.1.** $N_v$: The set of one-hop neighbors of $v$

**Definition 1.2.** $N_v^2$: The set of two-hop neighbors of $v$

**Definition 1.3.** $C_v$: $\{v\} \cup N_v^2$

**Definition 1.4.** $C_{n(v)}$: $N_v$

**Definition 1.5.** Partial Cover Dependency Graph of $v$: a graph $H(C,F)$ such that

1. $C = \{C_v, C_{n(v)}\} \cup \{C_u, C_{n(u)}\} \forall u \in N_v$
2. $\exists f(c_1, c_2) \in F \iff \exists u \in V | u \in c_1 \land u \in c_2$

After constructing $H$, each cover is assigned a weight and a degree. The weight of a cover is defined as the sum of the weight of the vertices in that cover, and the degree is defined by the number of edges for that cover. Figure 3 shows a graph and the corresponding partial cover dependency graph of a vertex in that graph.

2) PCDG Algorithm: The PCDG algorithm uses 2-hop information for immediate setup of the graph, as described in the previous section. After initial setup, the algorithm no longer updates any information beyond 1-hop. Figure 4 shows the automaton for PCDG.

IV. SIMULATION SOFTWARE

V. EXPERIMENTS

Experiments were conducted to test algorithm performance and examine the relationship between maximizing network lifetime and minimizing vertex cover.
A. Minimum Weighted Vertex Cover

For the MWVC problem, we tested the DGMM algorithm against a similar algorithm developed in [8]. The Koufogiannakis/Young algorithm uses a similar coin flipping mechanism between vertices, with each one choosing to be a ‘root’ or a ‘leaf’ node, and the algorithm proceeds in a way that guarantees that two adjacent nodes making independent decisions will reach the same conclusions.

B. Network Lifetime

A key issue in developing algorithms in the Dependency Graph framework is the ranking of covers and the establishment of degrees. We chose a relatively simple method of ranking covers which has given good results. Most interesting, however, is that the initial ranking of covers seems to be superior to all subsequent rankings. This was determined during the experiment phase of our research, which is detailed in the next section.

Redundancy removal provides a tool to circumvent this problem. As each sensor reaches the end of its battery life, it can tell its neighbors to turn on. These neighbors can then negotiate with their neighbors, with redundant sensors turning off. The advantage of this approach is two-fold. First, there are no global reshuffle rounds. Communication costs are only incurred when strictly necessary to maintain the network. Second, sensors which are not affected by a particular event—those that are three or more hops away from a dying sensor—do not incur any extra cost as a result of a specific event.

Depending on the deployment details of a given network, communication costs may be much higher than sensing costs, so using redundancy checking as a means of network maintenance may extend network lifetimes. In section ??, we explore this potential through simulation.

We chose the DEEPS algorithm developed in [2] for target coverage and modified it suitably for vertex cover. This is a state-of-the-art two-hop algorithm which has been demonstrated through simulation and real-world experiments to improve network lifetimes. DEEPS insures network coverage by assigning targets to sensor nodes, preferring the strongest member of weakest sets to take charge.

DEEPS requires global reshuffles to maintain coverage, and those shuffles are proactive, they take place on a schedule rather than on an as needed basis.
C. Experimental Design

Random connected graphs were constructed, with the number of nodes and edges as the inputs. Nodes received a random weight between 400 and 1000. Graph construction proceeded by a modification of Erlang’s method: all possible edges were generated and then random edges were chosen until the desired number of edges had been added to the graph. In order to ensure connectivity, Spanning Trees were constructed for each graph and connected together until each graph was a connected graph.

We developed a discrete-event simulator in Ruby. Ruby was chosen primarily because of its strong system of multiple inheritance and the relative ease of unit testing and debugging. This allowed us to easily construct new families of nodes that used common simulation algorithms with very little repetition. In its current state, once a new type of node is defined, the network construction and simulation for both MWVC and Network lifetime require only a few lines. The node definitions can generally be written in two methods describing the state machine and communication behavior of the node type.

The source code for each algorithm and the simulation framework are open source and available for download.³

For the MWVC problem, graphs were constructed with 120, 240, 480, and 960 vertices with average degrees of 3, 6, 12, 24, 48, and 96. 50 random graphs were generated at each size, and the Koufogiannakis/Young distributed algorithm⁴, our distributed modification of the Gonzalez Generalized Maximal Matching Algorithm⁵, and versions of each with the redundancy checking algorithm were run on each generated graph, with results being averaged.

For the Network Lifetime problem, the difficulty was to capture the communication cost associated with running the covering algorithms. We assume that in every case, a constant amount of energy is required to maintain the sensing and information sharing functions of the network. In our simulation, this cost is only applied to sensors that are “on” in a given round. The cost of re-organizing the sensors is a global cost, as every sensor in the network is required to participate in establishing a new vertex cover. This is applied as a constant drain on all sensors in the network. We conservatively simulate this drain as being on a spectrum from free to being equal to the cost of the information sharing and sensing function of the network.

We tested PCDG and DEEPS in two scenarios: one where each algorithm performs a global reshuffle in each round, and one where each algorithm sets up an initial cover, and then uses redundancy checking to perform local maintenance on an as needed basis. Graphs were constructed with 20, 40, and 80 vertices and average degree of 3, 6, and 12. 25 experiments were run for each graph size.

D. Experimental Results

1) Minimum Weighted Vertex Cover: As expected, the addition of the constant time redundancy check improved results for both the DGMM algorithm and the K/Y algorithm. Figures 5 and 6 show the improvement. In our experiments the affect was small on average, less than 10%, but this could be viewed as significant given the low cost of the routine. More surprising was the difference in communication rounds between K/Y and DGMM. DGMM consistently resolved in one-tenth to one-third the number of communication rounds required by K/Y. Figure 7 shows the difference in communication rounds required. The main reason for the difference is that in every communication round, DGMM is guaranteed to resolve each edge connected to one of a given node-pair in each round. The number of unresolved edges in the graph therefore quickly dwindles.

The quality of solutions produced by K/Y and DGMM are similar. Figure 8 shows the quality of solutions

³Using the mercurial VCS, command hg clone https://rvertex.graphcomplexity.googlecode.com/hg/graphcomplexity-rvertex will retrieve a copy of the code repository.

This paper uses revision 67446e3ca7 of the code base
between the two algorithms. K/Y holds an edge but that edge is slight.

2) Network Lifetime: When communication cost for network maintenance is considered to be negligible (or ignored), DEEPS outperforms PCDG by about 10% in our simulations. Figure 9 shows the performance of both algorithms in a communication cost free setting. DEEPS with reshuffle also outperformed DEEPS with redundancy checking when maintenance costs were considered to be free.

PCDG using redundancy checking outperforms PCDG with global reshuffle regardless of the maintenance cost.

Figure 10 shows the performance of PCDG without maintenance costs.

When communication costs are accounted for, the performance of DEEPS and PCDG improves when local reorganization using redundancy checking is used in place of global reshuffling. Figures 11 and 12 show how performance is affected by accounting for network maintenance.

REFERENCES

Fig. 10: PCDG Without Communication Costs

Fig. 11: Deeps with Communication Costs

Fig. 12: PCDG with Communication Costs


